White Paper: An Efficient API and Binary Encoding for Distributed Performance Monitoring

1 Introduction

In order to understand and debug the performance characteristics of high-performance distributed applications, units of precisely time-stamped information, or "events", are collected from the distributed components of the application itself, the network links, and the middleware and operating system services used by the application components at each host. In order to reduce the work of analyzing the resulting data, the events are sent over the network to a central "collector" daemon. Thus, monitoring results can be processed and viewed on a single host without the need to go out onto the network and aggregate the various log files. The NetLogger Toolkit (http://www-didc.lbl.gov/NetLogger/) has been built around this basic methodology of precision time-stamps and automatic aggregation.

For several reasons, minimizing both the number of bytes transmitted on the network and the number of CPU cycles used to transmit them can be crucial to the effectiveness of a monitoring system. First, the amount of time used creating and sending a single event limits the maximum granularity of instrumentation: if it takes N nanoseconds to send an event, the tightest loop that can be instrumented with minimal (1%) perturbation of the application is N*100 nanoseconds per iteration. The size of the events when transmitted over the network to the "collector" also limits the instrumentation, particularly in data-intensive applications where bursts of application activity cause bursts of both application data and monitoring data. In order to avoid exacerbation of precisely the types of network problems that are being debugged, the bandwidth consumed by the monitoring data should be at least one, and probably two, orders of magnitude less than the application's. Finally, the number of bytes used for each event determines the maximum flow of events that can be processed by a single collector daemon; if the daemon is overrun on a reliable (TCP) connection, the sender (presumably an application component) will eventually stall while writing the monitoring information. Although systems like Kangaroo (http://www.cs.wisc.edu/condor/kangaroo/) could help with receiver-limited systems, this remains an important limitation, especially for real-time monitoring and analysis.

Monitoring and application instrumentation data have two important characteristics that conflict with the desire for complete efficiency. First, the types of events are dynamic, both within one sender, and from one sender to the next. Thus the number of bytes for an event is variable and, in practical terms, the application cannot simply "compile in" the memory layout of the messages, as would be done for a fixed binary protocol. Second, the monitoring events themselves may be transient; this is particularly true for application instrumentation, where successive debugging iterations may add, remove, or change many of the events generated by identical runs of the application. This requires that the overhead of adding or removing events in the code be very low – hopefully no higher than the language's *print* statement. Unfortunately, these factors prevented the use of very powerful and efficient binary I/O libraries such as PBIO

(http://www.cc.gatech.edu/systems/projects/PBIO/), which require that a separate C *struct* be declared and compiled into the application for each type of event.

In this white paper, we present a binary encoding for dynamic event types that attempts to minimize sender CPU cycles and the amount of bytes transmitted on the network. This encoding has been incorporated into the NetLogger Toolkit, which provides APIs in eight languages: C, C++, Fortran77, Fortran90, Java, Perl, Python, and TCL.

2 Terminology

The term "event" has already been defined in passing as a unit of precisely time-stamped information. It should be noted that the more common definition of "event" as an indication of a change in state does *not* necessarily apply; the criteria for event generation are application-dependent.

In this paper, the process that writes the formatted bytes for an event onto the network will be called the "sender", and the host that process lives on will be called the "sending host". Similarly, the event collector will be called the "receiver" and the event collector's host, the "receiving host".

3 Event Model

The model of an "event" used here is very simple: an identifier, a time-stamp, and zero or more name, type, and value tuples, called "fields". In all cases, the 'name' is an array of octets that is not further interpreted, thus allowing Unicode or numeric values as well as ASCII strings. The 'identifier' provides a sender-unique identifier for the event. The 'type' indicates the data type of the value; so far Integer, Float, and String are supported. Zero or more of the fields may be designated as 'constants', that is, their values will not change between events with the same identifier. The order and type of the non-constant fields also does not change between events with the same identifier.

In the encoding described below, many size limitations were imposed on the event model, mostly for the sake of efficiency. This was done on the assumption that a single event will commonly be used to transmit only a few (non-constant) numeric or string values. In our experience, this is true for most application instrumentation and many types of system monitoring. A summary of the limitations is shown below.

Event Model Element	Limitation
Total size, header + body	128K
Total number of fields	255
Maximum length of any field	255 bytes
Maximum length of any field name	255 bytes
Timestamp	8 bytes
Number of basic data types	256

Given these limitations, it would be impossible to represent even a medium-sized time series within a single event; instead each set of measurements for the time series would have to be transmitted separately. The tradeoff is that small events, such as counters, block identifiers, and “number of bytes so far”, are very efficient.

Note that the event model here is tied to the idea of a sender and receiver, and in particular the identifier must only be unique to a given sender. This means that if events from multiple senders are mixed together, additionally identifying information is needed to distinguish between their respective “unique” identifiers.

4 Binary Event Encoding

The event is formatted in two parts: the first is called the ‘header’ and the second is called the ‘body’. For a given sender, the header is written once per event identifier, and the body is written for every event. Because the number of distinct event identifiers is expected to be quite small compared to the total number of events sent, efficient routines for formatting and sending the body are more important than for the header.

This section will describe the data type (common to the header and body), header, and body formats.

4.1 Data Type Formats

There are only three data types defined so far: INT, FLOAT, and STRING. The capitalized names will be used to distinguish these from data types of the same name in programming languages.

- INT – A 32-bit signed integer
- FLOAT – A 4-byte signed floating point number in standard IEEE format.
- STRING – A one-byte length followed by 0 to 255 octets

The data is always sent in the sender’s byte order, a style of data transmission known as “receiver makes right”. This provides maximum efficiency for the sender at the cost of some added complexity (and, to a lesser extent, lost efficiency) at the receiver.

The first planned extensions to these types are unsigned 4-byte integer and floating point numbers, both signed and unsigned 8-byte integer and floating-point numbers, and signed and unsigned 2-byte integers.

4.2 Header Encoding

The header consists of a length, a minor and major version number, an identifier, an architecture code, the number of fields and constant fields, a list of field names and types, and finally the constant values for the constant fields. The exact byte layout is shown in Table 1.

Table 1: Header Encoding

Byte	0	1	2		3	4	5	6	7
Contents	Length		**1**	major version	Minor version	Identifier			
Byte	8	9	10	11 .. 11+N		11+N .. 11+M+N			
Contents	arch	# of fields	# of consts	Field names		Constant values			

- Length – The length of the entire header, including the 2 bytes for the length itself.
- Major version – The first bit of the major version is used as a "header" flag, so it is always '1' in the header. The number formed by the other 7 bits starts counting at 1, with 0 reserved for experimental versions.
- Minor version – Just an 8-bit (unsigned) number
- Identifier – 32-bit unsigned integer
- Arch – A code for (sending) architecture type. This is a bitwise OR of flags. Currently the only flag is big/little endian.
- # of Fields – The total number of fields, including constants that will be in this and every successive body message
- # of Consts – The number of fields that have constant values
- Field Names – The name (a STRING) of each field. The constant fields are always listed before the non-constant ones, and in the same order as the Constant Values.
- Constant Values – The type and value of each constant, in the same order as the Field Names.

The header does not include a timestamp because the header and the first body are sent at almost the same time, and the body (see below) has a timestamp already.

4.3 Body Encoding

The body consists of a length, a major and minor version, an identifier, a timestamp, and the values for the non-constant fields declared in the header.

Table 2: Body Encoding

Byte	0	1	2		3	4	5	6	7
Contents	Length		**0**	major version	Minor version	Identifier			

Byte	8	9	10	11	12	13	14	15
Contents	time-stamp, seconds				time-stamp, fractional seconds			
Byte	16 .. 16+N							
Contents	Field values							

- Length – The length of the entire header, including the 2 bytes for the length itself.
- Major version – The first bit of the major version is used as a "header" flag, so it is always '0' in the body. The number formed by the other 7 bits starts counting at 1, with 0 reserved for experimental versions.
- Minor version – Just an 8-bit (unsigned) number
- Identifier – 32-bit unsigned integer
- Time-stamp, seconds – Same as integer part of NTP time-stamp
- Time-stamp, fractional seconds – Same as fractional part of NTP time-stamp
- Field values – One byte indicating a type followed by the value (the length of the value is part of the type itself, as this information is not needed for fixed-length types).

5 Binary Event API

The binary event encoding was implemented in C, and therefore the NetLogger C API was used as a basis for the API. The main reason that C++ was not chosen is that C by virtue of being a lowest common denominator is more easily mapped to a variety of other languages, allowing APIs in both script languages (Perl, Python, and TCL) and Fortran. In addition, the type of low-level coding needed to get maximum speed would probably use mostly C idioms (*memcpy*, *&* operator, **p++*, etc) anyways. The current plan is to implement the Java API as a Java Native Interface (JNI) wrapper around the C code, although a "pure" Java API is also a possibility.

Although the general strategy of "wrapping" and using a C library does create architecture dependence, it has two advantages. First, there is only one body of code that defines the encoding, which makes maintenance easier. Second, if done correctly the mapping between the languages can take advantage of the inherent speed of the underlying C code.

5.1 C API

The heart of the C NetLogger API, and the only place where significant changes were made, is the *NetLoggerWrite()* call:

```
NetLoggerWrite(handle, “event-name”, “Const1=3
Const2=Hello”, “Var1=%d Var2=%f Var3=%s”, v1, v2, v3)
```

Note that the format string for both the constants and variables still looks like ULM. In order to make this call efficient, several things happen under the hood:

1. The event name is looked up in an internal hash table. If that lookup succeeds, steps 2 and 3 are skipped.
2. A new message object is allocated, and the constant values are copied into it. Then the format string is parsed and space is allocated for all the variable values (255 bytes is allocated for each string), and their types are recorded as simple one-byte codes.
3. This message object is put into the internal hash table using the event name as a key.
4. The values are copied into the existing message object
5. The message object is ‘serialized’ into an output buffer

The result of this procedure is that on successive calls to NetLoggerWrite, the constant and variable format strings do not need to be parsed. In addition, all memory is allocated on the first call, and subsequent calls only need to copy from the user’s values into the waiting space.

5.2 Other languages

Note: This section is not complete, pending the completion of the binary APIs in the “other” languages.

Because the C API uses the *varargs* library to simulate what in most high-level languages would be done with a temporary list object, the mapping from the script languages, and particularly Java and (non-*varargs*) C++, will require some cleverness. The current approach of passing a pre-formatted string will not work well, e.g. in Python:

```
NetLogger.write( “event-name”,
     “Const1=3 Const2=Hello Var1=%d Var2=%f Var3=%s” %
     (v1,v2,v3))
```

Instead, something closer to the C API will have to be used, with a language-specific construct replacing the *varargs*. Code in the C library will translate all the language-specific C mappings to a common routine.

The result from this work should be a significantly faster API in almost all languages (Java being the possible exception).

6 Performance Comparisons

Note: This section is not complete, pending the completion of the binary APIs in the "other" languages.

This section compares the performance of different binary API's, both with each other and with the ULM encoding.

7 Conclusions

This white paper presented a binary encoding for logging events over the network. Basic design choices were discussed briefly, and some implementation details related to the APIs were discussed. Efficiency, both of the code itself and of the programmer time needed to create and maintain it, were the guiding design principles. Due to this approach, many valuable types of data, such as multi-dimensional arrays, structs, and long strings, do not have a simple or natural mapping to this encoding. However, in our experience the vast majority of application and system monitoring can be easily translated to a relatively small (less than 30) number of simple values for each time-stamp. Whether this is true in general remains to be seen, but at any rate the binary encoding presented here should provide a good basis for evaluating other possible binary encodings of event information.